Ant Local Search for Combinatorial Optimization

Zufferey, Nicolas

How to cite

ZUFFEREY, Nicolas. Ant Local Search for Combinatorial Optimization. In: Bio-Inspired Models of Network, Information, and Computing Systems: 7th International ICST Conference, BIONETICS 2012 (Lugano, Switzerland, Dec. 2012). Berlin : Springer, 2014. p. 233–236. (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering) doi: 10.1007/978-3-319-06944-9_16

This publication URL: https://archive-ouverte.unige.ch//unige:47395
Publication DOI: 10.1007/978-3-319-06944-9_16

Ant Local Search for Combinatorial Optimization

Nicolas Zufferey[1]

HEC - University of Geneva, 1211 Geneva 4, Switzerland
nicolas.zufferey-hec@unige.ch

Abstract. In ant algorithms, each individual ant makes decisions according to the *greedy force* (short term profit) and the *trail system* based on the history of the search (information provided by other ants). Usually, each ant is a *constructive* process, which starts from scratch and builds step by step a complete solution of the considered problem. In contrast, in *Ant Local Search* (ALS), each ant is a local search, which starts from an initial solution and tries to improve it iteratively. In this paper are presented and discussed successful adaptations of ALS to different combinatorial optimization problems: graph coloring, a refueling problem in a railway network, and a job scheduling problem.

Key words: ant algorithms, local search, combinatorial optimization

1 Introduction

As exposed in [15], most ant algorithms are population based methods where at each *generation*, a set of ants provide solutions, and at the end of each generation, a central memory (the trail system) is updated. The role of each ant is to build a solution step by step from scratch. At each step, an ant adds an element to the current partial solution. Each *decision* or *move* m is based on two ingredients: the *greedy force* $GF(m)$ (short term profit for the considered ant, also called the *heuristic information*) and the *trail* $Tr(m)$ (information obtained from other ants). Let M be the set of all possible decisions. The probability $p_i(m)$ that ant i chooses decision m is given by

$$p_i(m) = \frac{GF(m)^\alpha \cdot Tr(m)^\beta}{\sum_{m' \in M_i(adm)} GF(m')^\alpha \cdot Tr(m')^\beta} \quad (1)$$

where α and β are parameters, and $M_i(adm)$ is the set of admissible decisions that ant i can make. When each ant of the population has built a solution, the trails are generally updated as follows: $Tr(m) = \rho \cdot Tr(m) + \Delta Tr(m)$, $\forall m \in M$, where $\rho \in]0,1[$ is a parameter representing the evaporation of the trails (often

fixed to 0.9 or a close value), and $\Delta Tr(m)$ is a term which reinforces the trails left on decision m by the ant population of the current generation. That quantity is usually proportional to the number of times the ants selected decision m, and to the quality of the obtained solutions when decision m was made. More precisely, let N be the number of ants, then $\Delta Tr(m) = \sum_{i=1}^{N} \Delta Tr_i(m)$, where $\Delta Tr_i(m)$ is proportional to the quality of the solution provided by ant i if it has selected decision m. Overviews of ant algorithms (including *ant colony optimization*) are [3, 5].

Often, in order to get competitive results, it is unavoidable to apply a local search method to the solutions provided by such *constructive* ants [6]. In contrast, as proposed in [15], a more important role can be given to each ant by considering each of them as a *local search*, where at each step, as in every ant algorithm, the considered ant makes a decision (i.e. performs a move) according to the greedy force and the trail. The resulting method is called *Ant Local Search* (ALS).

The paper is organized as follows. In Section 2 are briefly described the main elements of a local search and the ALS methodology. Then are presented successful adaptations of ALS to three combinatorial optimization problems, namely the graph coloring problem (Section 3), a refueling problem in a railway network (Section 4), and a job scheduling problem with setup, tardiness and abandon issues (Section 5). The paper ends up with a conclusion in Section 6.

The contribution of this paper is the following:

- some advantages of the ALS approach are accurately highlighted;
- it is showed that ALS is a flexible method, as it can be easily adapted to very different combinatorial optimization problems;
- a new ALS algorithm is proposed for the job scheduling problem (Section 5);
- guidelines are given to efficiently design a trail system within an ALS framework.

2 Ant Local Search (ALS)

A *local search* can be described as follows. Let f be an objective function which has to be minimized. At each step, a *neighbor* solution s' is generated from the current solution s by performing a specific modification on s, called a *move*. Let $N(s)$ denote the set of neighbor solutions of s. First, a local search needs an initial solution s_0 as input. Then, the algorithm generates a sequence of solutions $s_1, s_2, \ldots$ in the search space such that $s_{r+1} \in N(s_r)$. The process is stopped for example when an optimal solution is found (if it is known), or when a fixed number of iterations have been performed. Some famous local search algorithms are: the descent method (where at each step the best move is performed, and the process stops when a local optimum is reached), simulated annealing, variable

neighborhood search, and tabu search. In tabu search for example, when a move is performed from a current solution s_r to a neighbor solution $s_{r+1} \in N(s_r)$, it is forbidden (with some exceptions) to perform the inverse of that move during *tab* (parameter) iterations: such forbidden moves are called *tabu* moves. The solution s_{r+1} is computed as $s_{r+1} = \arg\min_{s \in N'(s_r)} f(s)$, where $N'(s)$ is a subset of $N(s)$ containing solutions which can be obtained from s by performing a non tabu move. Many variants and extensions of tabu search can be found for example in [7].

The ALS method is summarized in Algorithm 1, where N is the number of used ants. A *generation* consists in performing steps (1) and (2).

Algorithm 1 ALS

While a time limit is not reached, do

1. **for** $i = 1$ **to** N: apply the local search associated with ant i, and let s_i be the resulting solution;
2. update the trails by the use of a subset of $\{s_1, \ldots, s_N\}$.

In most ant algorithms, it is very time consuming to make a single decision according to Equation (1). For this reason, a quick way to *select* a move, based on the greedy forces and the trails, is proposed in [15] and described below (the advantages of such a selection process are more deeply discussed here). At each iteration of the local search associated with the considered ant, let A be the set of moves with the largest greedy force (resp. trail) values. Then, the selected move is the one in A with the largest trail (resp. greedy force) value (ties are broken randomly). Of course, this process is only interesting if $|A| > 1$, otherwise the trails (resp. greedy forces) will have no impact on the search. Such a way of selecting each move at each iteration leads to several advantages over most classical ant algorithms. More precisely, it is not required anymore to:

- compute the trails (resp. greedy forces) of *all* possible moves, as only $|A|$ computations are required (note that even if the trails can be stored in a matrix which is only updated at the end of each generation, the computation of the trail of a move often needs additional specific computation, as illustrated in the three next sections);
- *normalize* the greedy forces and the trails of the possible moves (without normalization, during the search, the range of the trail values might become much larger than the range of the greedy force values, which makes the search difficult to control);
- compute the probability $p_i(m)$ associated with each possible move m;
- consider and tune the parameters α and β (as the use of Equation (1) is avoided).

In other words, in contrast with most other ant algorithms, the greedy forces and the trails are *successively* used to make a decision (instead of *jointly*). Therefore, a significant amount of computing time is saved, and the tuning phase of the algorithm is reduced.

3 ALS for Graph Coloring

3.1 Presentation of the problem

Given a graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$ and edge set E, the *graph coloring problem* (GCP) [21] consists in assigning an integer (called *color*) in $\{1, 2, 3, \ldots, n\}$ to every vertex such that two adjacent vertices have different colors, while minimizing the number of used colors. The *k-coloring problem* (k-GCP) consists in assigning a color in $\{1, \ldots, k\}$ to every vertex such that two adjacent vertices have different colors. Thus, the GCP consists in finding a k-coloring with the smallest possible k. The GCP is usually tackled by solving a series of k-GCP's, starting with a large value of k (which is at most n) and decreasing k by one unit each time a k-coloring is found. In such a case, a solution is often represented by a partition of the vertices into k color classes, *conflicts* are allowed (i.e. adjacent vertices can have the same color), and the goal consists in minimizing the number of conflicts (if it reaches zero, a k-coloring is found and the algorithm stops). Many (meta)heuristics were proposed to solve the GCP and the k-GCP. For a recent survey, the reader is referred to [12]. As discussed in [21], there mainly exists three types of ant algorithms for the GCP. First and as in most of the cases, an ant can be a *constructive* heuristic [4]. Second, an ant can be a very simple *agent* which helps to make a minor decision [8]. Third, an ant can be a refined *local search* such as tabu search [15]. Even if the role of an ant can be defined in various ways, each decision is always based on the *greedy force*, which is associated with the self-adaptation of each ant, and the *trail system*, which represents the collaboration between the ants.

3.2 Adaptation of ALS

The ALS coloring method proposed in [15] is derived from *PartialCol* [2], an efficient tabu search algorithm for the k-GCP, where *partial legal k-colorings* are considered, which are defined as conflict-free k-colorings of a subset of vertices of G. Such colorings are represented by a partition of the vertex set into $k + 1$ subsets $V_1, \ldots, V_{k+1}$, where $V_1, \ldots, V_k$ are k disjoint color classes without any conflict, and V_{k+1} is the set of non colored vertices. V_c (with $c \leq k$) actually represents the set of vertices with color c. The objective is to minimize $|V_{k+1}|$ (if it reaches zero, a k-coloring is found and the algorithms stops). A neighbor

solution can be obtained from the current solution by moving a vertex v from V_{k+1} to a color class V_c (with $c \leq k$, which means that vertex v gets color c), and by moving to V_{k+1} each vertex in V_c that is in conflict with v (such vertices are thus uncolored). Such a move m is denoted $m = (v \rightarrow V_c)$. When it is performed, it is then tabu to move v back to V_{k+1} (i.e. to remove the color c from vertex v) for a few iterations.

In ALS for the k-GCP, an ant is a tabu search procedure derived from *PartialCol*. The greedy force $GF(m)$ of a move $m = (v \rightarrow V_c)$ is defined as the inverse of the number of adjacent vertices to v that are in color class V_c (if it is zero, $GF(m)$ is set to an arbitrary large number, because there is then no need to remove the color of other vertices). The trail value $Tr(m)$ associated with move m is defined as follows. Let x and y be two vertices, and let $s_i = (V_1, \ldots, V_k; V_{k+1})$ be a solution provided by a single ant i of the population at a specific generation. If ant i gives the same color c to x and y in solution s_i (i.e. $x, y \in V_c \neq V_{k+1}$), such an information should be transmitted to the ants of the next generations, and this information should be more important if x and y are in a large color class. During the search, a non colored vertex $v \in V_{k+1}$ is likely to move to a color class V_c containing vertices with which v is used to have the same color.

Formally, let

$$\Delta Tr_i(x, y) = \begin{cases} |V_c|^2 & \text{if } x \text{ and } y \text{ have the same colour } c \text{ in } s_i; \\ 0 & \text{if } x \text{ and } y \text{ have different colours in } s_i. \end{cases}$$

At the end of each generation and as in many classical ant algorithm, the trails are globally updated as follows: $Tr(x, y) = 0.9 \cdot Tr(x, y) + \Delta Tr(x, y)$, where $\Delta Tr(x, y) = \sum_{i=1}^{N} \Delta Tr_i(x, y)$. Finally, the trail of a single move $m = (v \rightarrow V_c)$ is defined as $Tr(v \rightarrow V_c) = \sum_{x \in V_c} Tr(v, x)$.

3.3 Results

Considering a set of 14 well-known and difficult benchmark instances (see `http://www.info.univ-angers.fr/pub/porumbel/graphs/`), below is a representative numerical comparison for the following coloring algorithms:

- CAS [4], a *Constructive Ant System* where each ant is a constructive procedure (as in most classical ant algorithms);
- ADS [8], an *Ant Decision System* where each ant can help to color a vertex;
- ALS [15], the already discussed *Ant Local Search*;
- *PartialCol* [2], the tabu search from which ALS was derived;
- *Mem* [11], a memetic algorithm which can be considered as the best coloring method, as it provides the best average results on each of the benchmark instances.

A time limit of one hour and the same computer were used for *PartialCol* and the three ant coloring algorithms (i.e. CAS, ADS and ALS). The experimental conditions of Mem were different (e.g., a time limit of five hours [11]). The reader interested in accurate and detailed comparisons of coloring methods is referred to [21] for ant algorithms, and to [11] for other state-of-the-art algorithms. The *density* $d \in [0, 1]$ of a graph is the average number of edges between two vertices. The results are summarized in Table 1. For each graph (first column) is mentioned its number n of vertices (second column) and its density d (third column). In the next columns are given the minimum number of colors used by each method to generate conflict-free colorings. Obviously, ALS performs much better than the other ant coloring methods. Secondly, it is clear that the ingredients added to *PartialCol* to derive ALS are useful. Finally, in contrast with ALS, CAS and ADS are not competitive with the best coloring methods. The main reasons are probably the following: in CAS and ADS, too many ingredients are *simultaneously* handled in order to make the decision of changing the color of a single vertex, and these ingredients are of different *natures* and should not be mixed together. On the contrary, ALS only manipulates, *successively*, a few ingredients.

Graph	n	d	CAS	ADS	ALS	$PartialCol$	Mem
$DSJC500.1$	500	0.1	17	15	12	12	12
$DSJC500.5$	500	0.5	68	56	48	50	48
$DSJC500.9$	500	0.9	167	135	127	127	126
$DSJC1000.1$	1000	0.1	29	25	20	21	20
$DSJC1000.5$	1000	0.5	122	104	86	89	83
$DSJC1000.9$	1000	0.9	313	255	225	226	223
$flat300\_28\_0$	300	0.48	43	36	29	28	29
$flat1000\_50\_0$	1000	0.49	120	101	50	50	50
$flat1000\_60\_0$	1000	0.49	121	102	60	60	60
$flat1000\_76\_0$	1000	0.49	120	103	85	88	82
$le450\_15c$	450	0.17	28	18	15	15	15
$le450\_15d$	450	0.17	28	18	15	15	15
$le450\_25c$	450	0.17	33	29	26	27	25
$le450\_25d$	450	0.17	33	29	26	27	25

Table 1. Comparisons between ant and state-of-the-art coloring algorithms

4 ALS for a Refueling Problem in a Railway Network

4.1 Presentation of the problem

The problem proposed in the 2010 INFORMS optimization competition consists in optimizing the refueling costs of a fleet of locomotives over a railway network [9]. It is assumed that there is only one source of fuel: fueling trucks, located at yards. A solution of the problem has two important components [16]: (1) choose the number of trucks contracted at each yard, and (2) determine the refueling plan of each locomotive (i.e. the quantity of fuel that must be dispensed into each locomotive at every yard). Such components are respectively called the *truck assignment problem* (TAP) and the *fuel distribution problem* (FDP).

The constraints are the following: the capacity of the tank of each locomotive is limited, as well as the maximum amount of fuel a truck can provide the same day; a locomotive cannot be refueled at its destination yard; there is a maximum number of times (which is two) a train can stop to be refueled (excluding the origin); it is forbidden to run out of fuel. The encountered costs are the weekly operating cost of each fueling truck, the fuel price per gallon associated with each yard, and the fixed cost associated with each refueling. The problem consists in finding a feasible solution minimizing the total costs. A detailed description is provided in [9] as well as a literature review (recent papers in that field are [10, 13, 19]).

A *stop* is defined with a triplet (locomotive, yard, day). If a stop is *open*, it means that the involved locomotive can get fuel at that yard on that day (for the considered realistic instance, the same locomotive cannot stop two times at the same yard during the same day). The stop is *closed* otherwise (fuel distribution is not allowed). A *solution* of the problem can be modeled with a pair (T, S), where T and S are vectors of respective sizes equal to the number of yards and the total number of stops (among all locomotives). Component j of T is the number of contracted trucks at yard j, and component i of S indicates if stop i is open or closed. As proposed in [16], if a solution of the TAP is provided (i.e. if T is known), the corresponding FDP can be quickly and optimally solved with a *flow* algorithm. For the flow algorithm, all the stops for which there are trucks on the associated yards are initially open. Then the provided flow solution will indicate which stop will be actually used (and the non used stops will be closed). This also means that a solution of the TAP can be evaluated by the use of the flow algorithm. Thus, only the TAP is considered below.

4.2 Adaptation of ALS

First, a descent algorithm for the TAP can be easily designed as follows. From a current solution (T, S), a move consists in adding a contracted truck to a yard

(*add move*), or in removing a contracted truck from a yard (*drop move*). When a move is performed, the associated FDP is optimally solved and evaluated by the flow algorithm. During the evaluation, all the costs are considered: the refueling costs for the used gallons of fuel, the fixed refueling costs and the contracting costs of the trucks. The resulting descent method for the TAP (denoted DTAP) is proposed in Algorithm 2, with the setting $|Y^{(add)}| = |Y^{(drop)}| = 5$ (other settings were tested but did not improve the results).

Algorithm 2 DTAP: Descent for the Truck Assignment Problem

Construct an initial solution (T, S).

While a local optimum is not reached, do:

1. in solution (T, S), randomly choose a set $Y^{(drop)}$ containing yards for which drop moves are allowed; for any yard $y \in Y^{(drop)}$ and from (T, S), remove a truck from it, and apply the *flow algorithm* to evaluate such a drop candidate move;
2. in solution (T, S), randomly choose a set $Y^{(add)}$ of yards; for any yard $y \in Y^{(add)}$ and from (T, S), add a truck to it, and apply the *flow algorithm* to evaluate such an add candidate move;
3. from (T, S), perform the best move among the $| Y^{(drop)} \cup Y^{(add)} |$ above candidate moves, and rename the resulting solution as (T, S).

An ALS method for the TAP can be derived from the above descent algorithm [17], where each ant is a procedure identical to DTAP, but with a *learning process* based on a trail system, which is defined as follows. Let x and y be two yards. The trail $Tr(x, y)$ associated with yards x and y aims to indicate if it is a good idea to have trucks on both yards x and y in the same solution. At the end of the current generation, such trails are globally updated as follows: $Tr(x, y) = 0.9 \cdot Tr(x, y) + \Delta Tr(x, y)$, where $\Delta Tr(x, y)$ is the number of trucks on x and y, computed only for solutions of the current generation having trucks on both x and y. A move can be denoted by $(x \rightarrow s)$, indicating that a truck is added to or removed from yard x, in the current solution s handled by the considered ant. Let $Tr(x \rightarrow s)$ be its associated trail value. It is straightforward to set $Tr(x \rightarrow s) = \sum_{y \in s} Tr(x, y)$ if it is used as follows. If $(x \rightarrow s)$ is an add move, among the possible add moves, it is interesting to select a move with a large $Tr(x \rightarrow s)$ value (because the history of the search seems to indicate that having trucks on yard x, as well as on the yards which already contain trucks in the current solution s, is a good idea). On the contrary, if $(x \rightarrow s)$ is a drop move, among the possible drop moves, it is better to select a move with a small $Tr(x \rightarrow s)$ value.

The greedy force $GF(x \rightarrow s)$ of a move $(x \rightarrow s)$ is simply the resulting objective function value. The way that an ant selects a move at each iteration is now described, according to the greedy force and the trail system. Remember that in DTAP, the performed move is the best among the ones in the set $| Y^{(drop)} \cup$

$Y^{(add)}$ |, with $|Y^{(add)}| = |Y^{(drop)}| = 5$. For the descent algorithm associated with an ant of ALS, two sets $T^{(add)}$ and $T^{(drop)}$ of size ten are first randomly chosen (other sizes were tested but without leading to a better performance). Then, let $Y^{(add)}$ (resp. $Y^{(drop)}$) be the subset of $T^{(add)}$ (resp. $T^{(drop)}$) containing the five moves with the best trail values (which is much quicker to compute than the objective function values, as the use of the flow algorithm is avoided). The performed move among $Y^{(add)} \cup Y^{(drop)}$ is the best one according to the greedy force. Therefore, for DTAP as well as for the descent algorithm associated with an ant, the performed move has the best objective function value among a sample of ten evaluated solutions. This will allow to better measure the impact of the trail system on the search.

4.3 Results

The algorithms were tested on an *iMac* (3.06 GHz Intel Core 2 Duo processor, 4 Go 1,067 MHz RAM) with a time limit of 120 minutes. In order to fairly compare ALS and DTAP, the latter is restarted from scratch each time a local optimum is found (as long as the time limit is not reached). The considered realistic instance, proposed in [9], is characterized as follows: 73 yards, 213 trains, 214 locomotives, and a planning horizon of 14 days.

DTAP and ALS can be compared in Table 2. In the second column is given the average value (in $, over ten runs) of DTAP for the corresponding execution time (indicated in the first column). In the third column is indicated the average gain (in $, over ten runs) of ALS over DTAP. In addition, Figure 1 compares the evolution of the best encountered solution value (average over 10 runs, on the vertical axis) during 120 minutes (7200 seconds, on the horizontal axis). One can easily deduce that the learning process (i.e. the trail system) introduced to DTAP to derive ALS is relevant as it leads to non negligible savings.

The INFORMS contest involved 31 research teams. The approaches of the three best teams can be found in [9]. The winners of the contest (Kaspi and Raviv) formulated the problem as a MILP (mixed integer linear program). They found a lower bound $LB = 11,399,670.58$\$ and their best result was 0.30\$ above LB. Thus the gap between ALS and LB is approximately 0.35%. Note that a major advantage of ALS over the MILP proposed by Kaspi and Raviv is its *flexibility*: it can be easily adapted if non linear components are added to the problem. For example, one can assume that the weekly operating cost of the fueling trucks might be concave with the number of trucks located at the same yard.

Time (min)	DTAP	Gain(ALS)
0	11,605,703	0
15	11,455,921	16,584
30	11,450,137	16,178
45	11,445,918	12,502
60	11,442,113	9,039
90	11,439,601	6,636
105	11,439,601	7,919
120	11,439,587	7,929

Table 2. Gain of ALS over DTAP (in $)

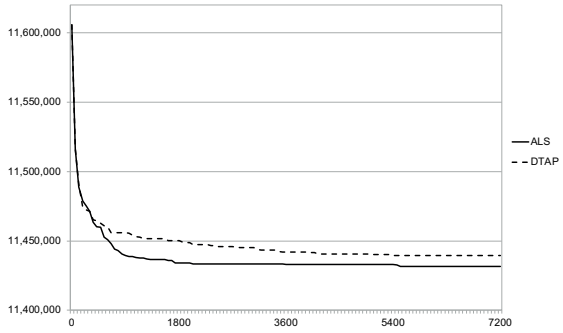

Fig. 1. Evolution of ALS and DTAP

5 ALS for a Scheduling Problem with Abandon Costs

5.1 Presentation of the problem

Consider a scheduling problem (P) where a set of n jobs have to be performed on a single machine. It is possible to abandon (reject) a job j, and in such a case, an *abandon* cost u_j is encountered (it can represent that j is allocated to an external resource). For each job j are known: its *processing* time p_j, its *release* date r_j (it is not possible to start j before that date), its *due* date d_j (the preferred completion time of job j) and its *deadline* d'_j (the latest allowed completion time of job j), such that $r_j \leq d_j \leq d'_j$. For each job, its starting time B_j (or equivalently its completion time C_j because $C_j = B_j + p_j$) has to be determined.

On the one hand, a job cannot be started before its release date (i.e. $B_j \geq r_j$), because one can assume that it is not possible to get the raw material associated with j before that date. On the other end, a job cannot be finished after its deadline (i.e. $C_j \leq d'_j$), because one can assume that the client will refuse j after

that date. If $C_j \in]d_j, d'_j]$, j is said to be *late* (from the client perspective) and this will be penalized in the objective function by a component $f_j(C_j)$.

In addition, various families of jobs are considered, assuming that jobs with comparable characteristics belong to the same family or *product type*. If two jobs j and j' of different families are consecutively performed, a setup cost $c_{jj'}$ is encountered and a setup time $s_{jj'}$ has to be taken into account.

A solution s can be represented as a vector of size n where component j contains the value of B_j. If job j is unperformed, one can put a fictitious -1. Let $U(s)$ (resp. $I(s)$) be the set of unperformed (resp. performed) jobs of solution s. The goal consists in minimizing an objective function with three components: (1) the abandon costs $\sum_{j \in U(s)} u_j$; (2) the setup costs $\sum_{j \rightarrow j'} c_{jj'}$ of the consecutively performed jobs $j \rightarrow j'$ of $I(s)$; (3) the penalty costs for late completion times $\sum_{j \in I(s)} f_j(C_j)$, where $f_j(C_j)$ is a *regular* (i.e. non decreasing) function depending on C_j. In the literature, the two most popular regular objective functions are the sum of completion times (i.e. $\sum_j C_j$) and the sum of tardiness (i.e. $\sum_j T_j$, where $T_j = \max\{0; C_j - d_j\}$), as well as their weighted versions (i.e. $\sum_j w_j \cdot T_j$ and $\sum_j w_j \cdot C_j$).

Problem (P) was first proposed in [1] where the authors proposed a branch and bound algorithm able to tackle most instances with up to 30 jobs. Other relevant references in the field are [14, 20]. Up to date, the only existing (meta)heuristic for (P) is a tabu search proposed in [18], denoted $Tabu(P)$, which is based on four types of moves: *reinsert* a job (i.e. remove a job $j \in I(s)$ from its current position in s and insert it somewhere else in s), *swap* two jobs of $I(s)$, *add* a job (i.e. move a job from $U(s)$ to $I(s)$), and *drop* a job (i.e. move a job from $I(s)$ to $U(s)$). At each iteration, a random sample (tuned to 15%) of the four types of move is generated, and the best move among the sample is performed. Note that if a moves leads to a non feasible solution (e.g., if the processing of some jobs overlap in time, or if a job is finished after its deadline), a repairing process is used, which allows to shift or drop jobs (see [18] for details on the repairing process).

5.2 Adaptation of ALS

The ALS method for (P) is denoted $ALS(P)$ and is summarized in Algorithm 3, where $N = 20$ and $I = 500$ were used in the experiments (other settings were tested but did not lead to better results). The local search operator of the initialization phase is $Tabu(P)$. The specificities of Algorithm 3, when compared to Algorithm 1, are the following: the trail matrix Tr is initialized before the main loop, Tr also appears in the construction operator, and Tr is updated as soon as an ant provides a solution (denoted s' in the pseudo-code).

The *greedy force* GF of a move is simply its associated objective function value. The *construction operator* starts from an empty solution s. Then, at each step, a non considered job j is scheduled within s (or rejected if it is better). More precisely, let $(j \rightarrow p, s)$ be the move consisting in inserting job j at position p of the jobs sequence of solution s (when inserting a job, it is allowed to shift, or to reject, some already scheduled jobs if it can make the solution feasible or less costly). At each step, among the q (parameter tuned to 25) less costly insertions $(j \rightarrow p, s)$, choose the one with the largest associated trail value $Tr(j \rightarrow p, s)$, which is defined as

$$Tr(j \rightarrow p, s) = \sum_{x \text{ before } p \text{ in } s} Tr(x, j) + \sum_{x \text{ after } p \text{ in } s} Tr(j, x)$$

The *trail* matrix $Tr(x, y)$ associated with the scheduling of job x before job y is based on the number of times x was positioned before y during the search (and on the quality of the associated solutions). More precisely, at the end of each generation, the trail matrix is updated as follows: $Tr(x, y) = 0.9 \cdot Tr(x, y) + 1/\hat{f}$, where $\hat{f}$ is the average value of the solutions of the current generation for which job x is scheduled before job y.

The local search operator of the main loop is exactly $Tabu(P)$, with the following modification. Every ten iterations, it is imposed to perform an add move based on Tr and GF as follows. Among the q (parameter tuned to 25) moves of type $(j \rightarrow p, s)$ with the largest trail value $Tr(j \rightarrow p, s)$, perform the less costly insertion (i.e. the one with the largest GF value).

Algorithm 3 Algorithm $ALS(P)$

Initialization of the trail matrix Tr

1. generate randomly N solutions and improve each of them with a local search operator during I iterations;
2. initialize the trail matrix Tr with the N improved solutions.

While a time limit is not reached, do:

1. construction operator: build a solution s based on the trail matrix Tr and the greedy force GF;
2. improve s by the use of a local search operator during I iterations (based on Tr and GF) and let s' be the resulting solution;
3. use s' to update the trail matrix Tr.

5.3 Results

$Tabu(P)$ and $ALS(P)$ were tested on a computer with processor Intel i7 Quandcore (2.93 GHz RAM 8 Go DDR3) during $30 \cdot n$ seconds (i.e. about four hours if

$n = 500$ jobs). Note that $Tabu(P)$ is restarted every $I = 500$ iterations as long as the time limit is not reached, so that it can be fairly compared to $ALS(P)$, which uses each ant during $I = 500$ iterations. The instances are the same as the ones with setups described in [1] and the cost component $f_j(C_j)$ is $w_j \cdot T_j$. In Table 3 are compared the upper bounds $UB(P)$ provided by the branch and bound algorithm of [1], and the average gain (over 10 runs) of $Tabu(P)$ and $ALS(P)$ over $UB(P)$. One can remark that the average gain of $ALS(P)$ is more than 4,000 above the average gain of $Tabu(P)$: it is thus worthy to use the proposed trail system.

Instance	n	UB	$Gain(TS)$	$Gain(ALS)$
$STC_NCOS_01.csv$	8	920	220	220
$STC_NCOS_01a.csv$	8	1,010	400	400
$STC_NCOS_15.csv$	30	22,321	4,710	4,710
$STC_NCOS_15a.csv$	30	6,449	865	865
$STC_NCOS_31.csv$	75	6,615	-250	-250
$STC_NCOS_31a.csv$	75	7,590	-250	-250
$STC_NCOS_32.csv$	75	25,774	1,554	1795
$STC_NCOS_32a.csv$	75	16,908	110	110
$STC_NCOS_41.csv$	90	85,378	40,740	42,175
$STC_NCOS_41a.csv$	90	26,828	7,731	8,246
$STC_NCOS_51.csv$	200	308,770	169,095	164,559
$STC_NCOS_51a.csv$	200	318,740	107,770	170,510
$STC_NCOS_61.csv$	500	1,495,045	0	-463
$STC_NCOS_61a.csv$	500	1,821,085	6,480	6,480
Averages		**295,960**	**24,227**	**28,508**

Table 3. Comparison of $Tabu(P)$, $ALS(P)$ and an upper bound $UB(P)$

6 Discussion and Conclusion

Within the ant algorithms field, paper [21] was a first try to answer the question: *"What should be the role of a single ant?"*. In most ant algorithms, an ant is a constructive heuristic. In contrast, an ant is a local search in ALS. Even if the role of an ant can be defined in various ways, each decision is always based on the greedy force (representing the self-adaptation of each ant), and the trail system (modeling the collaboration between the ants). This paper shows that ALS is a promising algorithm for combinatorial problems: it obtained competitive results for three very different combinatorial optimization problems (graph coloring, a refueling problem, and a job scheduling problem). For the graph coloring problem, it was numerically showed that ALS performs much better than a standard

ant algorithm. Therefore, a straightforward avenue of research would be to adapt ALS to other combinatorial problems.

Another important issue is indirectly tackled in this paper: *"How should be defined an efficient trail system?"*. For the three considered problems, it would not be relevant to respectively transmit the following *myopic* (or very local) information to the ants of the next generations: (1) a pair (vertex x, color c), as two colorings can be equivalent if the color indexes are permuted; (2) a pair (yard y, truck t), as yard y might be empty if trucks are located in a yard close to y; (3) a pair (position p, job j), as the scheduling of a job strongly depends on the scheduling of the other jobs (especially if setups are considered). In contrast, the above proposed trail systems are respectively based on: (1) the assignment of the *same color* to some vertices; (2) the assignment of trucks to a *same set* of yards; (3) the *relative order* in which jobs appear. In other words, a trail system should *globally cover* specific characteristics of the considered problem.

References

1. Ph. Baptiste and C. Le Pape. Scheduling a single machine to minimize a regular objective function under setup constraints. *Discrete Optimization*, 2:83 – 99, 2005.
2. I. Bloechliger and N. Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & Operations Research*, 35:960 – 975, 2008.
3. C. Blum. Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews*, 2(4):353–373, 2005.
4. D. Costa and A. Hertz. Ants can colour graphs. *Journal of the Operational Research Society*, 48:295–305, 1997.
5. M. Dorigo, M. Birattari, and T. Stuetzle. Ant colony optimization – artificial ants as a computational intelligence technique. *IEEE Computational Intelligence Magazine*, 1 (4):28–39, 2006.
6. M. Dorigo and T. Stuetzle. *Handbook of Metaheuristics*, volume 57, chapter The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances, pages 251–285. In F. Glover and G. Kochenberger (Eds), 2003.
7. F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.
8. A. Hertz and N. Zufferey. A New Ant Colony Algorithm for Graph Coloring. In Pelta and Krasnogor, editors, *Proceedings of the Workshop on Nature Inspired Cooperative Strategies for Optimization, NICSO 2006, June 29–30*, pages 51–60, Granada, Spain, 2006.
9. INFORMS RAS Competition. http://www.informs.org/community/ras/problem-solving-competition/2010-ras-competition. 2010.
10. M. Kuby and S. Lim. The flow-refueling location problem for alternative-fuel vehicles. *Socio-Economic Planning Sciences*, 39 (2):125 – 145, 2005.
11. Z. Lu and J.-K. Hao. A memetic algorithm for graph coloring. *European Journal of Operational Research*, 203:241 – 250, 2010.
12. E. Malaguti and P. Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17 (1):1 – 34, 2010.

13. S. M. Nourbakhsh and Y. Ouyang. Optimal fueling strategies for locomotive fleets in railroad networks. *Transportation Research Part B*, 44 (8-9):1104 – 1114, 2010.
14. C. Oguz, F. S. Salman, and Z. B. Yalcin. Order acceptance and scheduling decisions in make-to-order systems. *International Journal of Production Economics*, 125:200 – 2011, 2010.
15. M. Plumettaz, D. Schindl, and N. Zufferey. Ant local search and its efficient adaptation to graph colouring. *Journal of the Operational Research Society*, 61:819 – 826, 2010.
16. D. Schindl and N. Zufferey. A local search for refueling locomotives. In *Proceedings of the 54th annual conference of the Administrative Science Association of Canada - Production & Operations Management Division (ASAC 2011)*, pages 53 – 61, Montreal, Canada, July 2 – 5 2011.
17. D. Schindl and N. Zufferey. Ant Local Search for Fuel Supply of Trains in America. In *Proceedings of the 1st International Conference on Logistics Operations Management*, Le Havre, France, October 17 – 19 2012.
18. S. Thevenin, N. Zufferey, and M. Widmer. Tabu search to minimize regular objective functions for a single machine scheduling problem with rejected jobs, setups and time windows. In *Proceedings of the 9th International Conference on Modeling, Optimization & Simulation (MOSIM 2012)*, Bordeaux, France, June 6 – 8 2012.
19. B. Vaidyanathan, R. K. Ahuja, J. Liu, and L. A. Shughart. Real-life locomotive planning: New formulations and computational results. *Transportation Research Part B*, 42(2):147 – 168, 2008.
20. B. Yang and J. Geunes. A single resource scheduling problem with job-selection flexibility, tardiness costs and controllable processing times. *Computers & Industrial Engineering*, 53:420 – 432, 2007.
21. N. Zufferey. Optimization by ant algorithms: Possible roles for an individual ant. *Optimization Letters*, to appear (DOI: 10.1007/s11590-011-0327-x), 2011.